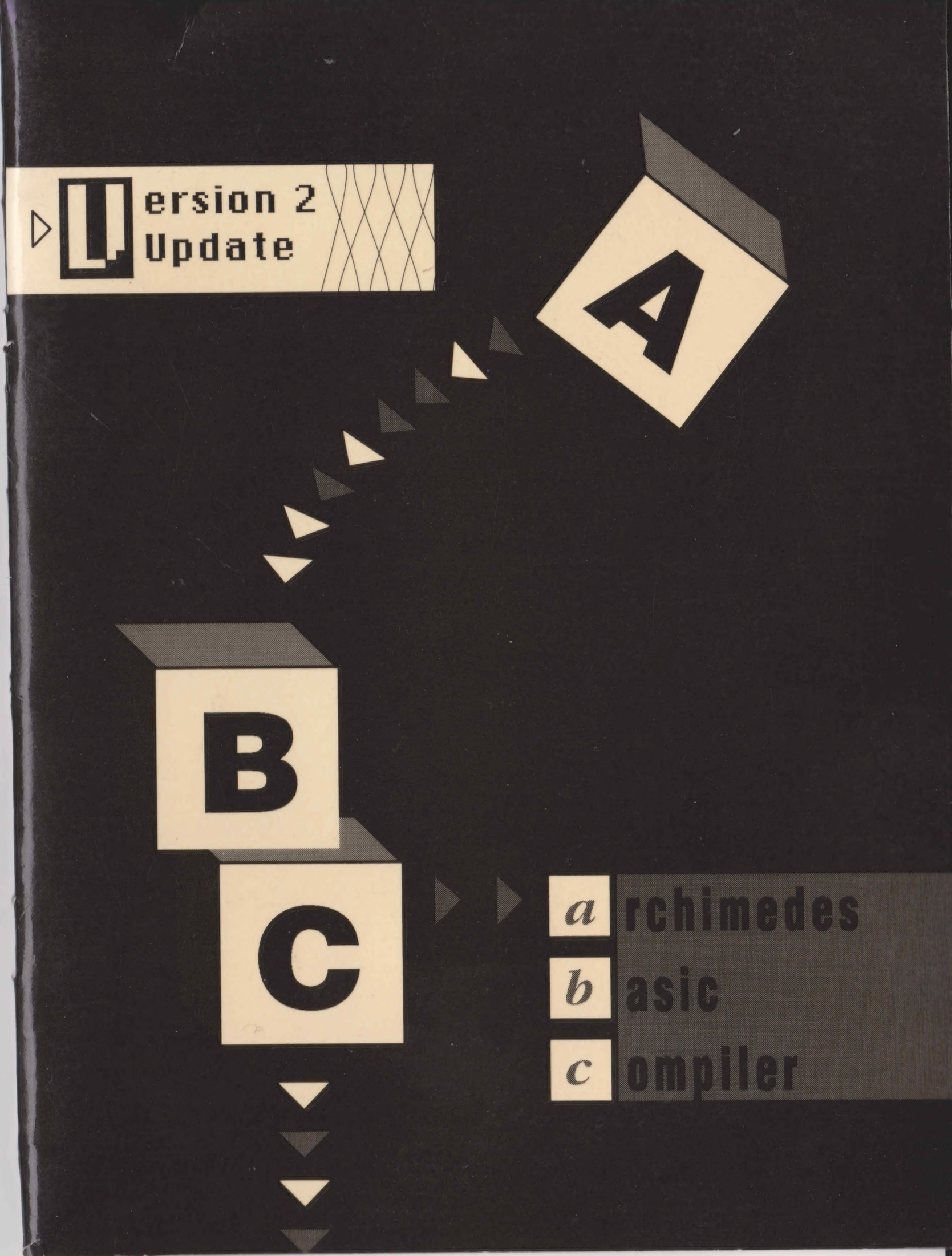
Version 2
Update
A
B
C
archimedes
basic
compiler

Archimedes Basic Compiler

Version 2 Update User Guide

Paul Fellows

DABS
PRESS

Archimedes Basic Compiler
Version 2 Update User Guide

ISBN 1-870336-76-3.
First edition April 1989

Editor: David Atherton
Typesetting: David Tomlinson
Additional material: David Atherton, David Tee, Andrew Hutchings
Cover: Clare Atherton

Published by Dabs Press, 5 Victoria Lane, Whitefield, Manchester, M25 6AL
Telephone 061-766-8423, Telecom Gold 72:MAG11596, Prestel 942876210.

Typeset in 10 on 11pt Palatino by Dabs Press using the Acornsoft VIEW wordprocessor, MacAuthor, Apple Macintosh SE and LaserWriter II NT.

Printed in Great Britain by Bolton Print, Bolton, Lancs.

Contents

Acknowledgements

Paul Fellows would like to thank David Tee of Select Software for providing the 'Shell' enviroment which enables ABC to run under the RISC OS desktop.

Dabs Press would like to thank Colin Hutchinson and Neil Turner for their vigorous beta-testing of the compiler, and Andrew Hutchings for providing the additional demonstration programs.

Introduction

This addendum guide covers the improvements and additions to Version 2 of the Archimedes Basic Compiler and should be seen as a supplement to the ABC User and Reference Guides.

Double and Extended Precision Floating Point

Version 1.01 of ABC provided single-precision floating point variables via the Acorn Floating Point Emulator. This kept storage requirements to a minimum since each single-precision number only occupied four bytes of memory. However, for a number of applications it is desirable to have greater numerical accuracy than the six or so digits provided by single-precision. For this reason Version 2 of ABC gives access to three different forms of floating point variables. These are referred to as single- , double- and extended-precision.

The BASIC interpreter uses the presence of '$' and '%' characters to distinguish between floating point numbers, strings and integers. This scheme has been extended to control the different types of floating point through the use of the '`' and '&' characters.

Note: All references to the '`' character in this documentation are describing the use of the character whose ASCII value is &60 (96). This appears as the back-tick character when using the ISO fonts and will be produced by the back-tick key on the keyboard, unless the Archimedes has been reconfigured a different KEYBOARD or COUNTRY setting in which case it may appear as '£'.

If the name of a variable ends in the backtick character '`' the compiler will treat it as double-precision. Likewise, if the last character is '&' the variable is assumed to be extended-precision.

By default, the precision of a floating point variable is assumed to be single if its name ends in a letter (A..Z or a..z) , a digit (0..9) or the underscore character '_'.

Thus the following program is assumed to use two single-precision (hot and cold) and one double-precision (double`) variables.

```
10 REM > exampleFP
20 :
30 hot  =  27.453
40 cold = -10
50 :
60 double` = 4.6692016090
```

Under the BASIC interpreter this program will run although all three variables will be represented in BASIC's unique 5-byte format.

The rules which apply to the names of simple variables also apply to arrays thus:

```
10 DIM block`(1000)
```

will, by default, set up an array of 1000 double-precision numbers occupying 8000 bytes.

This does pose a compatibility problem with the use of '&' as a variable name suffix in that the BASIC interpreter will not allow it. Thus programs which use '&' to specify the use of extended precision cannot be tested under the interpreter. However, all is not lost! This problem can be worked around through the use of the TYPE compiler directive described later.

Accuracy and Range

Single precision numbers provide only very limited accuracy, typically about six decimal digits. Use of double precision increases this to about 15 while extended precision provides up to 19. Table 2.1 provides a comparison.

	Digits	Exponent Range	Storage
Single	6	+38 ... -44	4
Double	15	+308 ... -324	8
Extended	19	+4932 ... -4950	12

Table 2.1. Precision comparison.

@% and Print Formatting

To allow for the use of higher precision it has been necessary to change the meaning of the print control variable '@%'. Each of the four bytes now has a special meaning:

Byte Four – The Flags Byte

This controls the format in which PRINT# sends numbers to data files. It also controls whether STR$ obeys the remaining bytes. If the top bit is set STR$ obeys the print formatting bytes, otherwise it does not.

The default value of the flags byte is zero.

Byte Three – Significant Digits

The lower five bits of this byte specify the number of significant digits which are allowed. A number will be printed in such a way that the total number of significant digits is not more than this limit. Thus, if the number of digits before the decimal point plus the number of decimals specified by byte two of @% is less than or equal to this limit the number is printed in full using fixed-point format. Otherwise the number will be printed in exponential format with this number of significant digits.

Byte Two – Decimal Places

The lower five bits of this byte specify the number of decimal places which will be used for fixed format numbers.

Byte One – Field Width

This controls the number of character places in which the number will be output. If the number has fewer digits than this it will appear right-justified with the appropriate number of leading spaces. If the number won't fit in the field width it is printed in as few places as possible.

The top byte of the default value of @% is affected by the new TYPE Compiler Directive (see Section 4) in order to control the default format in which floating point numbers are output to data files.

The default value of @% always has its bottom three bytes set to &0A0A0A. This means that by default floating point numbers will be printed in general format using 10 significant digits in a field of 10 places.

@% and floating point in data files

Version 1.01 of ABC made use of a bit in @% to control the format in which floating point numbers were placed in data files. This scheme has been extended to use a second bit. Table 3.1 shows how the bits settings control the format.

Format	Bit-25	Bit-24
Single (4-Byte)	0	0
Acorn (5-Byte)	0	1
Double (8-Byte)	1	0
Extended (12-Byte)	1	1

Table 3.1. Bit settings

Thus to output numbers in extended precision set both bits in @% as follows.

```
1000 @% = @% OR &03000000
```

Compiled programs will read data files in any of these formats and carry out the necessary conversion automatically. This enables full compatibility with the interpreter. However if your compiled program produces data to be read by INPUT# in the interpreter, you'll have to stick to the Acorn format.

The format used to store floating point numbers in data files is as follows:

The Type Byte

Compiled programs will put one of &F0, &E0, &D0 or &80 in the type byte according to the value of @%.

&F0 {F}loating point, ie, single precision
&E0 {E}xtended
&D0 {D}ouble
&80 Acorn-format

The Number

The 4, 5, 8 or 12 bytes of the floating point number will be output directly to the file, lowest byte first.

Compiler Directives

Five new Compiler Directives have been implemented and these are outlined below, with the exception of REM {REGISTERS} which is covered 'The Assembler' section.

TYPE

ABC provides a directive to allow full control over the relationship between the type of a variable and the last character of its name – as discussed in section 1 of this Addendum Guide. This is the TYPE directive. In its simplest form it can be used to force the compiler to change the precision used for simple floating point variables. The following program shows this in use:

```
10 REM {TYPE = DOUBLE}
20 INPUT x
30 INPUT power
40 :
50 variable= x ^ power
60 :
70 PRINT variable
```

All three variables (x, power and variable) will be created as eight-byte double precision variables.

Note: The TYPE directive must be placed at the top of the program before any executable code.

A more complex version of the directive is available which allows you to specify the meaning of one of the special characters thus overwriting the default meanings. For example:

```
10 REM {TYPE ` = EXTENDED}
```

This tells the compiler to use extended-precision for all variables which end in the backtick character.

These directives can be used to get around the limitation imposed by the BASIC interpreter on the use of '&' to imply extended precision.

In most cases a program will probably only want to use one sort of floating point for all its variables. In which case the simplest solution is to avoid using the '&' suffix and to specify the same precision for variables with no suffix and for those with a '`'. For example:

```
10 REM {TYPE = EXTENDED}
20 REM {TYPE ` = EXTENDED}
```

The program can then be tested under the interpreter although the results will be rather less accurate than those of the compiled program!

If for some reason you need to use two forms of floating point within a program you can still maintain testability. Again you should avoid use of '&' and specify the precision for the other two possibilities. For example:

```
 10 REM {TYPE = SINGLE}
 20 REM {TYPE ` = EXTENDED}
 30
 40 DIM array(100000)
 50 ...
 .. ...
100 num`=array(I%)*array(J%)
120 ...
```

This program uses single precision for the array since 100000*4 bytes is a very large amount of store (400k). Using extended precision would require three times as much memory - 1.2Mbs! However, the product of two array elements is put into the extended-precision number 'num`'. This is useful since it is certain that the calculation will not overflow. An extended-precision variable is easily capable of holding the square of the largest possible single-precision number.

Forcing Integers

It also possible to force the compiler to treat variables which do not end in the % character as integers. For example:

```
 10 REM {TYPE = INTEGER}
 20 ...
 .. ...
100 fred = 10.3
```

```
110
120 PRINT fred
```

This will print '10' because variable 'fred' is an integer. Of course, the interpreter will still treat 'fred' as being floating point.

It is not possible to change the meaning of the '%' and '$' suffix characters.

FLOAT and NOFLOAT

In some applications it is useful to know that your program is free of all use of floating point so that the Acorn Floating Point Emulator need not be loaded. This is particularly relevant with published commercial software, as you cannot distribute Acorn's FPE code without written agreement from them.

The directive:

```
REM {NOFLOAT}
```

will cause the compiler to fault any attempt to use floating point arithmetic (or library routines which do so). Thus if the program compiles successfully with this directive set then it can be assumed that it does not use the Floating Point Emulator.

Warnings

The number of warnings is now counted for the whole compilation rather than only for the last pass. A pair of Compiler Directives are also provided which enable you to tell the compiler not to stop when a warning is produced.

```
REM {NOWARNINGS}
```

will cause the compiler to issue the warning without stopping, and

```
REM {WARNINGS}
```

reverses this. These directives may be used in pairs to mark particular sections of a program. The default action is to stop. This is useful in that several trivial errors can be fixed in one go without the need to recompile several times. Beware however of consequential errors. When you fix one problem, others often disappear.

Floating Point Indirection

By default, floating point indirection only transfers four bytes. However, if the TYPE Compiler Directive (see previous section) has been used to change the size of the simple variables to eight or 12 bytes, the floating point indirection operator will transfer this number instead.

It is possible to force the compiler to use one particular size and thus override the TYPE directive in the following way:

```
 10 REM {TYPE = DOUBLE}
 20 DIM block% 1000
 30 ..
 .. ..
100 |{S}block% = 37.5
```

The {S} after the | character forces the compiler to perform a single-precision (four-byte) transfer. {D} for double and {E} for extended precision operations may be used in a similar fashion.

The Assembler

A number of improvements have been made in this area, and these are outlined below.

Floating Point Constants

The EQUF assembler directive is now accompanied three new varieties:

EQUFS EQuateFloatingpointSingle
EQUFD EQuateFloatingpointDouble
EQUFE EQuateFloatingpointExtended

These place the value given to them in-line in the assembly code file to their respective precisions.

The behaviour of EQUF is governed by the setting of the TYPE Compiler Directive described in that section. EQUF will use the same precision as will be used by a simple variable.

Manifest Constants

It should be noted that all floating point manifest constants will be held to extended precision. This includes named manifest constants defined with DEF. The name of a manifest constant may contain the %,& or ` character but this has no significance.

CALL and USR

Version 2 of ABC now exports the values of A%,B%,...H% in registers when CALL or USR is used with only a single argument (the address).

If the CALL or USR is not in a procedure or function there is no ambiguity. However, when inside a PROC or FN, the compiler will first try to find LOCAL variables A%..H% belonging to that particular procedure or function. If the procedure or function does not have its own LOCAL A% etc., then the compiler will use the global assignments and issue a warning that it is doing so. This is because it cannot grant the CALL or USR access to the LOCAL variables of any other procedure or function. By drawing your attention to this the compiler is passing the buck to you to make sure the correct values are being passed! If you are unsure about this, read the scope rules in the ABC Reference Guide.

The extended syntax of CALL and USR is still available and provides an easier way to specify parameters.

Register Names in the Assembler - R0-R15/PC

If at the end of pass 1 of the compilation, the compiler finds that any of the names:

r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15 R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15 pc Pc pC PC

have been used without being defined it will issue a warning and give the option of continuing with the name defined as a manifest constant of the appropriate value for use in the assembler. Thus, it is not neccessary to define all the register names in a program.

An additional directive has been provided to allow the names to be declared all in one go. This is:

```
REM {REGISTERS}
```

It should be placed at the start of the program. The register names will then be declared as manifest constants. This may cause a clash with the use of variables within the program (which is why it is not the default action).

Lower Case Assembler Mnemonics

These are now supported.

Structure Rules

Version 2 of ABC no longer insists that there is only one ENDPROC for every DEFPROC and only one function return for every DEFFN. Some restrictions do however remain, for example the procedure:

```
1000 DEF PROCddle
1010 .....
1020 IF X=0 THEN 1050
1030 ....
1040 ENDPROC
1050 ....
1060 ENDPROC
```

is not allowed. This program must be converted to use the multi-line IF...THEN...ELSE...ENDIF.

The rules governing the use of ENDPROC in ABC programs are as follows:

a) The body of a procedure starts at the DEFPROC and lasts until:
   - an ENDPROC is found which is not nested within another structure.
   - the end of the program is reached.
   - another DEF is encountered.

b) An ENDPROC which is nested within some other structure will be compiled as a branch to the 'real' end of the procedure.

Thus the example above is illegal since neither of the ENDPROCs on lines 1040 or 1060 are nested within some other structure.

An example of legal structure is:

```
1000 DEF PROCddle1
1010 REPEAT
1020 IF X=0 THEN ENDPROC
1030 ...
1040 UNTIL FALSE
1050 ...
2000 DEF PROCddle2
2010 ...
2020 CASE X OF
2030 WHEN 1 : ENDPROC
2040 OTHERWISE
2045 ...
2050 ENDCASE
2060 ENDPROC
3000 DEF PROCddle3
3010 ...
3020 IF X=0 THEN
3030 ...
3040 ELSE
3050 ...
3060 ENDPROC
3070 ENDIF
3080 ...
3090 ENDPROC
```

The rules for the use of function returns are the same with one addition.

c) No function may return values of incompatible data types.

For example:

```
100 DEF FNx(A)
200 IF A = 0 THEN
210 = "HELLO"
220 ELSE
230 = PI/4
240 ENDIF
```

is not allowed. (It's not much use anyway because just about all you can do is print it!)

It is still sensible to use the one-to-one basis wherever possible to avoid ambiguity and to allow the compiler to check program structure fully.

The structure rules concerning FOR...NEXT and REPEAT..UNTIL etc remain intact.

RETURN Parameters

Version 2 of ABC now supports RETURN parameters. That is parameters to procedures or functions which return their final values back to the variables which were used when the call was made.

An example of the use of RETURN parameters:

```
 10 INPUT X
 20 INPUT Y
 30 ...
 40 PROCsort(X,Y)
 50 ...
 60 PRINT X
 70 PRINT Y
 .. ...
 99 END
100 DEF PROCsort( RETURN  A , RETURN B )
110 IF B>A THEN SWAP A,B
120 ENDPROC
```

The values of X and Y are copied into the variables A and B when the procedure is called. Inside the procedure, if B is greater than A then their values are exchanged. When the procedure returns, X is given the new value of A and Y the new value of B. Thus, the values of X and Y will be such that X >= Y.

The syntax for this is fully compatible with that of the interpreter although there is a minor restriction. This is that the actual parameter may be a simple variable or an array element but may not be an indirect expression.

Local Error Handling

Version 2 of ABC provides a full package of local error handling. However, it is not exactly as implemented by the interpreter. The specification provided by the compiler is as follows:

```
LOCAL ERROR <stmt>
ON ERROR LOCAL <stmt>
```

Both mean the same thing namely "set an error handler for this procedure". The error handler will be inherited at run time by any PROC or FN which is called from the scope of the local error handler.

```
 10 ON ERROR PRINT "Error detected"
 .. ...
 99 END
100 DEFPROCone
110 LOCAL ERROR PRINT "Local error detected"
... ...
150 ENDPROC
200 DEFPROCtwo
210 ERROR 1,"Crash!"
220 ENDPROC
```

If PROCtwo is called directly from the main program, then the error which it generates will be caught by the global error handler which is set up on line 10. Alternatively, if PROCtwo is called from PROCone, the local error handler set up by PROCone on line 110 will be activated. In a more complicated case there might be a third procedure, PROCthree, which made calls of PROCtwo and was itself called from either the main program or from PROCone. The error will be trapped by the global error handler if the sequence of calls has not been through PROCone. If PROCone has been called its local handler will trap the error.

Once a procedure returns to the place it was called from, any local error handler which it may have set up will be deactivated. Subsequent errors will then be trapped by the previously active error handler.

When an error occurs the stack is wound back to the level at which the error handler was set up and hence variables have their values restored. The interpreter does not do this, which can be very inconvenient.

```
LOCAL ERROR OFF
ON ERROR LOCAL OFF
RESTORE ERROR
```

These delete the record of the error handler at the current stack level. Using them inside a procedure which has no local error handler will have no effect.

Leaving a procedure (by ENDPROC) which set up a local error handler will automatically remove the handler, ie, ENDPROC automatically does RESTORE ERROR.

The effect of ON ERROR and ON ERROR OFF will be to set up the global error handler. The global error handler will only be called if either:

the program is at the global level

or alternatively:

no local error handler is in force

It must be noted that when an error occurs compiled programs still lose track of any stacked GOSUB/RETURN information. Use procedures instead!

Simple Integer Variables

The global integer variables A% to Z% and @% are always pre-declared. This should be transparent unless an attempt is made to define a manifest constant with the same name. For example:

```
DEF A% = 100
```

will cause an error.

ABClibrary

The ABC library is now provided as a module. It is automatically loaded in when the compiler is initialised via the ABC command file. This has reduced the size of the minimal program to about 2k.

All programs compiled with the new version of the compiler will need the ABCLibrary module loaded before they can be run, so turnkey applications will need to start with a *EXEC or *OBEY file, which RMLOADs the library first. (This is the recommended method under RISC OS)

Commercial programs developed with ABC may be distributed accompanied by a copy of the library free of any charge, subject to the usual acknowledgment.

Value of PI

The value of PI is stored as extended precision. It is:

3.1415926535897932 38

Internal Use of Extended precision

Independently of any Compiler Directives, internal calculations will be performed to extended precision and temporary results will be stored in this form to avoid loss of precision.

Using ABC from the Command line

It is now possible to use the compiler without going through its window-based user interface. To do so you simply give the names of both the source and object files on the command line. For example

```
*compile source.prog object.prog
```

The compiler will run in a text-only manner. Any error messages or warnings together with the prompts which they give will appear as plain text messages as and when necessary. The compiler directives LIST and NOLIST still control the output of the source listing.

There are several advantages to using the compiler in this way.

1. The compiler can be run in screen mode 0. Therefore the amount of memory set aside for the screen can be considerably reduced from 80K as used by the MODE 12 window interface. In fact MODE 0 only requires 20K but the configuration only takes effect in steps of 8K on the 300 series and 32K on the 440. Thus the minimum is either 24K or 32K. This can significantly increase the amount of memory which is available.

2. The compiler, when used in this way, does not require any system sprite memory. Thus it is possible to increase the amount of free memory by configuring the Spritesize to zero.

3. The speed of compilation will be increased by up to fifty percent. This is because in MODE 0 the video controller is not stealing as much time to access video memory and because the compiler is spend less time outputing the progress information.

4. This mode of operation allows the compiler to be run in a 'task window' under the new RISC OS operating system. Thus, compilation can carry on in the background whilst you use the machine for another purpose.

5. Through this mechanism you can use the command-line options which are detailed in the next section.

Command-line options

Several command-line options have been provided. These are specified to the compiler after the object program name on the command line. For example

```
*compile source.prog object -RUN
```

will cause the compiler to automatically run the object program once the compilation is over.

The options which are recognised are

-RUN

If this is specified the object code is run imediately after compilation. Otherwise the usual `Run object code (Y/N)` prompt is given.

-QUIT

This is the opposite of -RUN. If given the object code will not be run when the compilation is complete and the compiler quits without giving the prompt. This can be very useful when using an EXEC file to drive the compilation of a whole sequence of programs.

-DISC

This forces both the source and object programs to be dealt with to and from disc rather than in memory.

-NOLIST

This suppresses the listing of the source code. It overrides any `REM {LIST}` directives contained in the source program.

-HELP

This gives information about the version number of the compiler and help about the command syntax.

Improved HEAP and STACK directives

In order to improve the control over the use of memory a new form of the HEAP and STACK directives is now provided.

This takes the form

```
REM {HEAP% 90}
```

where the number specifies the percentage of the free memory which will be allocated to the heap.

Similarly

```
REM {STACK% 10}
```

would try to allocate 10% of the available memory to the stack.

Obviously the total cannot exceed 100% or an error will be given. These directives can be used in conjunction with the original absolute versions of the directives where appropriate. This is a useful method if you are attempting to compile under different memory configurations, perhaps on a mixed network of A310s, A440s etc.

Archimedes 305

There is insufficient memory on an Archimedes model 305 to run the full windowed ABC front end. If you are using a 305 you **must** use the command line method of compilation.

Using ABC with RISC OS

In addition to its use directly from the command line, ABC version 2 can be run under the RISC OS desktop.

On the ABC distribution disc, ABC is present in the !ABC application directory in a ready to run form. However, it is strongly recommended that you should install the system onto another floppy disc (or hard disc if you have one). You should then put the original disc away in a safe place.

Installing for RISC OS

If you are installing onto a floppy disc you must first have a blank-formatted disc. This will become your ABC work disc. Users with only a single drive should read the next section as well as this one before starting the installation in order to avoid unnecessary disc swapping.

To install ABC under RISC OS first enter the RISC OS Desktop. This can be done from BASIC or from the Supervisor prompt by typing

```
*Desktop
```

Place the ABC distribution disc in drive 0 and, using the mouse, click on the icon representing this drive. A directory viewer will appear showing the !ABC application directory, the !System directory, and some other files which are only important when installing for Arthur 1.2.

Using the mouse, open a directory viewer onto the target disc. If this is your hard disc, click on the hard disc icon on the icon bar. If it is a floppy disc, insert the floppy disc in any drive and click on the appropriate drive icon.

You should now have two directory viewers on the screen. One representing the ABC distribution disc and the other representing your target disc. Now, using the mouse, drag the !ABC and !System directory icons from the distribution directory to the target directory.

This will initiate the copying of all the files which you need. When the copy process has finished ABC is ready to run, from the target disc.

Icons, directory viewers and other desktop terms are explained in the RISC OS User Guide.

Note for Users with only a single drive

The copying of the directories from one floppy to another on a single drive machine will require a very large number of disc swaps. This can be avoided by using the RAM filing system. You should first set up the RAM disc to be large enough to hold all the files. This can be done from the desktop using the tasks window. Click the menu button on the Archimedes 'A' symbol and select the task display option from the menu which pops up. The tasks window will appear. Scroll it until you can see the entry for the RAM disc, then, using the mouse, drag the red bar adjacent to this entry to the right to increase the size of the RAM disc. 700K bytes will be required for the copy process.

Now open a directory viewer onto the RAM disc and perform the dragging operations described above to copy both !ABC and !System from the distribution disc to the RAM disc. Then insert the target disc and drag the two directories to the target directory.

Finally, reset the size of the RAM disc to its original value, after deleting any files which it contains.

Compiling a program

When the disc on which ABC has been installed is viewed from the Desktop Filer the ABC icon will appear. Double-clicking on this icon will bring up the ABC icon on the icon bar in the usual way. Basic source programs can then be compiled by dragging them onto this icon.

When a source program is dropped on the icon, a file requester for the object file will appear. You should either enter the full filename for the object file or drag the icon to a directory.

The usual listing and other messages will appear in a window as the compilation proceeds.

Unfortunately, you will not be able to do anything else whilst the compilation is being performed.

Running the object code

Once compiled your programs should appear in the Filer with the appropriate icon indicating an application or a module. Double clicking on these will run them as usual.

It is possible to make the compiler run the code as soon as compilation has finished. See the section on compiler options below.

If an error occurs

ABC will offer you the chance to enter the Basic Editor when it finds an error. You can do this safely even when running from the desktop despite the fact that the BASIC Editor is not a window-oriented program. If you do enter the editor and make some changes, save your program in the usual way and then leave the editor by pressing SHIFT-f4. This takes out out to the BASIC prompt. You should then leave BASIC by typing QUIT. This returns control to the Desktop manager and, after clicking the mouse once your desktop display will reappear exactly as it was.

Note that the Basic Editor is not present as a ROM module in RISC OS, so you should preload it before starting the compilation.

Using the Menu options

If you click on the ABC icon with the menu button, (the middle one on the mouse), a menu will pop-up. There are three options.

Info

Slide the mouse of the right of the menu as indicated by the arrow to obtain a box containing descriptive text about ABC including the version number.

Options

Slide off to the right to get a dialogue box from which the compiler options can be set. These are 'listing', 'RAM', and 'Auto Run' and are described in the next section.

Quit

Click to remove ABC from the icon bar.

Compiler options

The options window derived from the ABC menu allows you to control the action of the compiler.

1. The listing of the source text may be suppressed by de-selecting the 'listing' option.
2. The compiler can be told to perform a disc-to-disc (instead on RAM-to-RAM) compilation by deselecting the 'RAM' option.
3. The object code can be run automatically at the end of a sucessful compilation by selecting the 'Auto Run' option.

Bugs Fixed

Version 1 of ABC contained a number of bugs which have now been fixed in the new release. These are outlined below (if you know of or find any more please let us know!).

1. LOCAL ERROR

Programs containing:

```
LOCAL ERROR
```

caused the compiler to hang. This has now been eliminated.

2. Assembler Condition Code Synonyms

"HS" and "LO" are now supported as synonyms for "CS" and "CC" in the assembler.

3. FX4 State

The *FX4 state is reset on exit in all cases.

4. Compilation Time

The compilation time no longer includes the time taken to type in the file-names etc.

5. FIX

In ABC 1, LEFT$() MID$() and RIGHT$() on the left hand side of an assignment failed to FIX floating point numbers which were used as indices. This has been corrected.

Assignment to TIME failed to FIX floating point numbers. This has been corrected.

6. INPUT

INPUT#file with five byte format got the exponent wrong if bit three should have been set! This is now fixed.

7. SYS

SYS ... TO ; flags% accidently created a floating point variable whose name was ';'. Thus, when used in a service module the compiler objected on the grounds that floating point was not allowed! This no longer occurs. Also, unspecified register arguments and SYS calls are now set to zero.

8. TOP and LOMEM

The values of TOP and LOMEM were wrong. They are now correct.

9. Assembler

In the assembler, the ADR macros produced the wrong offset if the value was outside the range &00..&FF. It is now correct.

10. Module Memory

If the amount of module memory requested was not valid as ARM immediate data the compiler crashed with an internal error. This has been corrected.

11. FP Rounding

The rounding of floating point numbers during conversion to strings for output was incorrect in certain cases. Thus PRINT 91/7 gave 13.00000099. The number 13 is now printed.

12. File handling

This was a bug in the run-time library causing BPUT#file, A$ to fail. The stacking of R14 was not being done correctly in this routine. It now works as it should.

The runtime library now automatically coerces floating point or integers on input from a file, meaning that INPUT#file with mixed numerics now works correctly.

13. INPUT LINE

The runtime library has been corrected to take note of the flags passed in by the compiled code. Thus, INPUT LINE now acts correctly. Interestingly, READ LINE can now be used to read in entire lines from DATA statements.

14. NEXT A,B

This used to give an error. It is now allowed.

15. Falling into PROCs / FNs

If a program attempts to fall into the definition of a procedure (due to a missing END for example) an error will be given at run time. The compiler cannot trap this so it generates a branch-to-error just before the head of each PROC or FN body. The error message will be "Can't drop into PROC/FN" with error number 0 (ie a fatal error).

If you are utterly convinced that your program is not going to do this, you can tell the compiler to not bother with this trap using a new directive pair.

```
REM  { NOTRAPS }
```

turns off trap generation

```
REM { TRAPS }
```

turns them back on again.

16. SUM

The detokeniser now behaves itself with the keywords SUM and BEAT. This was because the jump table for these two keywords (which are in a group of their own) had become non word-aligned.

17. Assignment to TIME$

Assignment to TIME$ now supports all three forms of input string. Previously, the full date and time was allowed. It now also allows date-only and time-only as the interpreter does.

18. BY

If the word BY was not followed by a space version 1 failed to recognise it. Version 2 now correctly identifies it. Also, the compiler now allows FILL BY.

19. ELLIPSE with 5 parameters

ELLIPSE failed due to register R1 being corrupted if the last parameter was a complicated expression. A simple expression did not show the problem.

20. LET TIME = etc

A problem occured in the use of tokenised keywords following LET. These were incorrectly allowed through and created as FP variables. The compiler now gives a syntax error.

21. LEFT$ and RIGHT$

Interestingly enough, BASIC 4 and BASIC 5 do not do the same thing when presented with RIGHT$(A$,-1). One returns nothing and the other returns the whole string. ABC 2 is now compatible with the actions of BASIC 5. Previously it followed the BASIC 4 rule.

22. INPUT, VAL and READ

These now return a value of zero when presented with strings beginning with 'E'.

Other points to note

1. ABS is a floating point operation.

The documentation/read-me file should reflect this in the sections discussing service modules and/or the NOFLOAT directive.

2. Hex numbers

The compiler does not accept the lower case letters a, b, c, d, e and f for hexadecimal notation.

3. WHILE statements

The BASIC interpreter allows WHILE <expression> statement without a separating colon. The compiler insists on a valid separator.

4. Assembler

ABC deliberately gives the 'bad alignment' error if any attempt is made to place an instruction on a non-word boundary. It only does this if OPT is such that errors are to be reported. The BASIC interpreter does not do this, which we feel is a failing.

5. CASE statements

ABC insists that CASE <expression> OF <newline> be immediately followed by WHEN, OTHERWISE or ENDCASE. Nothing else is acceptable, not even comments.

Examples disc

The original examples disc has been enhanced with a suite of interesting programs from Andrew Hutchings of Derby, all of which were written with Version 1 of the Compiler and of course work with Version 2 as well, under both Arthur 1.2 and RISC OS.

DABS
PRESS